Going Vertical in Memory Management: Handling Multiplicity by Multi-policy

Lei Liu[1], Yong Li[2], Zehan Cui[1], Yungang Bao[1], Mingyu Chen[1], Chengyong Wu[1]
[1]State Key Laboratory of Computer Architecture, Institute of Computing Technology, Chinese Academy of Sciences
[2]Department of ECE, University of Pittsburgh
[1]{liulei2010, cuizehan, baoyg, cmy, cwu}@ict.ac.cn; [2]yol26@pitt.edu

Abstract

Many emerging applications from various domains often exhibit heterogeneous memory characteristics. When running in combination on parallel platforms, these applications present a daunting variety of workload behaviors that challenge the effectiveness of any memory allocation strategy. Prior partitioning-based or random memory allocation schemes typically manage only one level of the memory hierarchy and often target specific workloads.

To handle diverse and dynamically changing memory and cache allocation needs, we augment existing "horizontal" cache/DRAM bank partitioning with vertical partitioning and explore the resulting multi-policy space. We study the performance of these policies for over 2000 workloads and correlate the results with application characteristics via a data mining approach. Based on this correlation we derive several practical memory allocation rules that we integrate into a unified multi-policy framework to guide resources partitioning and coalescing for dynamic and diverse multi-programmed/threaded workloads. We implement our approach in Linux kernel 2.6.32 as a restructured page indexing system plus a series of kernel modules. Extensive experiments show that, in practice, our framework can select proper memory allocation policy and consistently outperforms the unmodified Linux kernel, achieving up to 11% performance gains compared to prior techniques.

1. Introduction

Efficient management of shared memory resources is important for application performance and system throughput. However, most existing memory and cache management mechanisms used in commodity production parallel machines adopt generic address-interleaving or scheduling/partitioning approaches that are oblivious to the diverse memory utilization characteristics and diverging resources requirements in today's heterogeneous environments. This often results in inter-program perturbation, resources thrashing, poor memory/cache utilization, and, consequently, degraded performance.

Several recent solutions attempt to segregate applications with different memory resources requirements by *horizontally* partitioning either main memory (DRAM banks) [10,16,17,29] or cache [15,24,30,31,32] into exclusive slices. These approaches avoid interference for programs with small memory footprints but might hamper performance of larger workloads by effectively reducing capacity. Partitioning and other memory allocation optimizations at the OS level are more flexible, and it performs well in many cases. For instance, given a multi-threaded workload, randomly interleaving pages to distribute each thread's accesses across different DRAM banks [18] could reduce row buffer conflicts, but does not address interference among applications in multi-programmed workloads. Therefore, it is desirable to design a memory management system that can choose appropriate allocation policies by distinguishing memory characteristics. To achieve this goal, simply integrating the best performing mechanisms is impractical as almost all state-of-the-art schemes requires expensive changes to memory controllers/allocators or cache hierarchies, not to mention the challenges in detecting and predicting the application requirements and conflicts.

To better meet the needs of diverse workloads and leverage the architecture advantages, we propose a software approach that simultaneously combines multiple allocation strategies at different levels of the memory hierarchy (e.g., DRAM bank level [10,16,17] and cache set level [8,15,26,31]). Enabling such *vertical* partitioning (through the DRAM banks and cache at the same time) creates a larger policy space from which the OS can choose. Additionally, we use random allocation when no partitioning method performs well. We integrate **H**orizontal partitioning, **V**ertical partitioning, and a **R**andomized paging policy (similar to Park et al.'s M^3 [18]) to create HVR, a low overhead, unified memory allocator that we implement in the Linux kernel. HVR automatically selects component policies dynamically based on different memory usage scenarios detected by an online application classification module that characterizes memory/cache behaviors.

To determine appropriate memory management policies and address conflicting allocation preferences, we perform more than 10,000 experiments for over 2000 workloads on production machines with mainstream processor and memory configurations. Based on a data mining approach to analyze our results, we generate a set of practical partitioning and coalescing rules together with a policy decision tree to help HVR with automatic policy selection, dynamic resources partitioning and coalescing. By combining these policies on the fly, HVR can handle diverse, complicated and dynamically changing memory allocation needs in daily computing and production environments where programs/jobs are launched and terminated arbitrarily. We implement HVR in around 3000 lines of source code in Linux kernel 2.6.32. We summarize our contributions below:

(1) Vertical partitioning (Section 3). Through a quantitative study we identify limitations of existing partitioning approaches. We leverage the overlapped bits

***ISCA'14*, June 14-18, 2014, Minneapolis, MN, USA.**
978-1-4799-4394-4/14/ $31.00(c)2014 IEEE.

(O-bits) in the physical page address for indexing both DRAM banks and cache sets to partition memory hierarchy vertically, and achieve accumulated gains from multiple horizontal partitioning methods.

(2) A multi-policy framework (Section 3). Based on O-bits, we design an efficient, flexible and all-in-one framework (HVR) that supports vertical partitioning, horizontal partitioning and random allocation scheme. HVR requires no hardware changes and performs well for both multi-threaded and multi-programmed workloads.

(3) A low-overhead, page-table-based cache-profiling module (Section 4). We develop an online module that dynamically captures and categorizes application cache utilization to assist appropriate memory allocation without offline profiling or expensive performance counters.

(4) Data mining driven allocation rules (Section 5). By adopting a data mining approach on extensive experimental results of various policies and numerous workloads, we derive a set of **partitioning and coalescing** rules used to appropriately partition resources while allowing non-interfering programs to live together for resource sharing.

(5) Real implementation in Linux kernel (Section 6). We restructure the buddy system and rebuild the physical page index in Linux kernel 2.6.32 to support all the component policies and the partitioning/coalescing rules in HVR. Dynamic application classification is implemented as kernel modules. Our implementation adds around 3000 lines of source code into the Linux kernel source tree.

Our experiments on a real machine show that vertical partitioning outperforms prior techniques [15,16]. Based on the classification module and partitioning/coalescing rules (90% accuracy verified by experimental results), HVR brings consistent performance gains to the unmodified Linux kernel and outperforms prior utility-based software partitioning [15,16] by up to 11%. HVR also achieves benefits for handling dynamic workload changes in real production environments.

2. Background

2.1 Buddy Memory Allocation System

Today's Linux operating system adopts a buddy system to manage and allocate physical pages to various applications in a low overhead and high efficient manner [13,23]. The current buddy system maintains 11 free lists with *orders* ranging from 0 to 10. The free list with order *R* organizes pages as blocks and each block has 2^R number of continuous pages. Upon a memory allocation request, the buddy system is responsible for identifying a free list with an appropriate order and selecting one block from the free list for allocation. One larger block with a higher order can be split into smaller blocks of lower orders when necessary. Buddy system aims to satisfy various memory requests from diverse applications as generally and efficiently as possible. To a considerable degree, the buddy system achieves the goal in the single-core era.

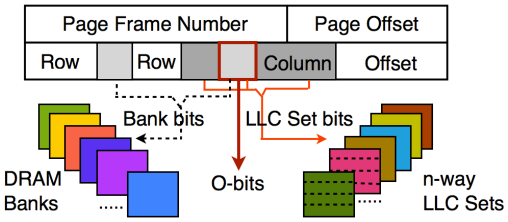


Figure 1. Address mapping for cache and bank partitioning.

2.2 Page-Coloring Based Memory Management

Multicore architecture poses new system design and optimization challenges, particularly on memory allocations, since it allows all applications to share LLC (Last Level Cache) and DRAM banks, resulting in severe contention in many cases. Previous research efforts [12,29] show that contention can significantly degrade the overall system performance and many solutions have been proposed to mitigate the contention problems.

One of the most effective optimizations is the page-coloring based software partitioning, which allows an OS kernel to leverage the underlying architecture information such as the physical address mapping of LLC and DRAM. With page-coloring, one can mitigate the contention problem [4,10,15,17,21,22,24,26] by modifying the kernel buddy system while avoiding expensive hardware changes to memory controllers or cache hierarchies.

Conventionally, there are two page-coloring based partitioning techniques, namely the cache partitioning and DRAM bank partitioning. As shown in Figure 1, cache partitioning can be achieved by using the bits in the OS physical page address that denote LLC set index (LLC color bits) as color bits. When allocating a page for an application, OS can assign a physical page with a specific color so that the application can only access the cache sets with the assigned color. Recent studies [10,16,17] utilize page-coloring to partition DRAM banks as there are also bits in the physical page address that denote DRAM bank address. The difference is that the bank color bits in physical page address might be distributed on some platforms. Figure 1 illustrates bank partitioning using page-coloring.

3 Memory Allocation Policies

In this section we study a number of existing allocation approaches and introduce our new methods to expand the memory allocation policy space. Based on a performance analysis of these policies we identify several memory allocation challenges and opportunities, which will be discussed in the later sections.

3.1 Horizontal and Vertical Partitioning

Traditional partitioning policies such as those mentioned in Section 2.2 are *horizontal* in that they partition either cache or main memory (DRAM banks) in one dimension. With the page-coloring technique the horizontal partitioning can be implemented by selecting bank or cache indexing bits as colors when allocating a page. Our detailed architectural

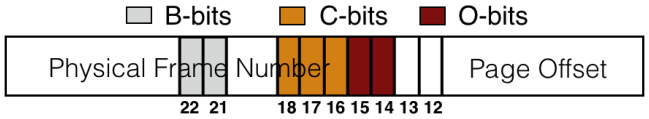


Figure 2. Address mapping from the view of OS and three categories of color bits on a typical multicore machine.

study reveals that the coloring bits can be classified into three categories: bank-only bits (B-bits), cache-only bits (C-bits) and overlapped bits (O-bits index both bank and cache in Figure 1). In particular, the O-bits enable a *vertical partitioning* (VP) that partitions both cache and memory banks, vertically through the memory hierarchy. Combining horizontal and vertical partitions forms a previously unstudied partitioning policy space. Assume, for example, there are L B-bits, M C-bits and N O-bits, we can use i B-bits, j C-bits and k O-bits to generate a partitioning policy represented as *(i,j,k),* where $0 \le i \le L$, $0 \le j \le M$ and $0 \le k \le N$ (value 0 is valid).

Figure 2 illustrates the three categories of coloring bits on a typical machine (Intel i7-860 with 8GB memory and 64 banks, B-bits: 21~22; C-bits: 16~18; O-bits: 14~15. An arbitrary coloring bit distribution depends on specific cache/memory configuration and can be detected by the approach presented in [16,18]). Actually, The 13 bit denotes bank, LLC and also L2 cache index, but we will not use it in partitioning since we do not want L2 cache to be partitioned. Table 1 lists six representative policies derived by these coloring bits. Each policy partitions certain resources (i.e., cache, memory or both) to a different extent and thus performs best in a certain scenario. For example, *A-VP* uses the two O-bits to partition both LLC and memory banks into four colors (groups) thus it is suitable for applications with modest memory/cache demands as only one fourth of the LLC and DRAM banks can be used with one color assigned.

3.2 Going Vertical?

Prior efforts [8,10,15,16,17,26,29,30,31] demonstrate that horizontal partitioning on memory or LLC is effective in eliminating inter-program interference and improving performance. With vertical partitioning and, more generally, our partitioning policy space, one important question is ***whether the benefits from the horizontal memory and cache partitioning can be accumulated (i.e., should we go vertical in partitioning?).***

To answer the above questions, we investigate over 200 random workloads composed of programs from SPECCPU 2006 [1]. This experiment includes two steps. In the first step, for each workload we run multiple experiments to obtain the performance gains of that workload on horizontal partitioning. The performance improvements are compared against the unmodified Linux kernel as the baseline. All of the experimental results are reported in the four-quadrant Cartesian plane in Figure 3. The horizontal axis and vertical axis represent the weighted speedup (see Section 7) improvements achieved through the Bank-only and Cache-only partitioning, respectively. Workloads that

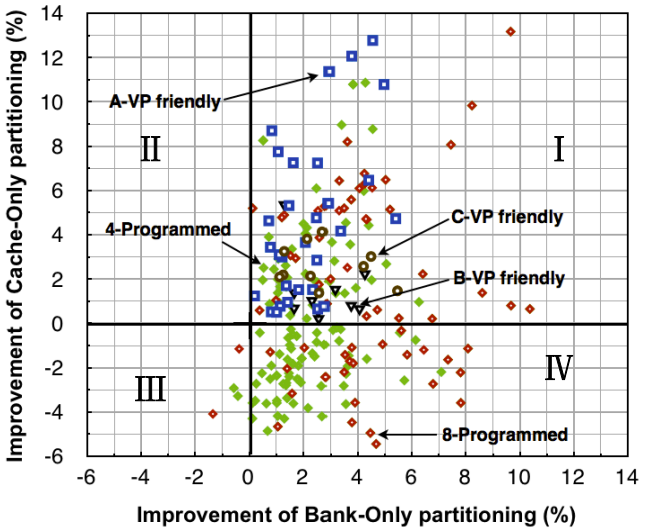


Figure 3. Performance improvement of Cache- and Bank-Only partitioning for 214 workloads. Green dots: 4-programmed; Red dots: 8-programmed; Blue squares: A-VP friendly; Black triangles: B-VP friendly; Brown circles: C-VP friendly. Note that the metric of overall system performance is Weighted Speedup which is defined in Section 7.1.

Policy	Coloring Bits	Description	Target Cores
Cache-Only	C-bits {16~18}	LLC → 8 groups	4/8-core
Bank-Only	B-bits {21~22}	Banks → 4 groups	4-core
Bank-Only	B-bits {21~22} O-bits {15}	LLC → 2 groups Banks → 8 groups	8-core
A-VP	O-bits {14~15}	LLC → 4 groups Banks → 4 groups	4-core
B-VP	B-bits {22} + O-bits {14~15}	LLC → 4 groups Banks → 8 groups	8-core
C-VP	C-bits {16} + O-bits {14~15}	LLC → 8 groups Banks → 4 groups	8-core

Table 1. Six representative partitioning policies.

contain 4 or 8 programs are denoted as 4/8-programmed workloads. From Figure 3 we can see about half of the tested workloads fall into Quadrant I. For these workloads, both Cache-Only policy and Bank-Only policy bring certain levels of performance improvements.

In the second step, we randomly select tens of workloads in the Quadrant I and use A, B and C-VP on them. We find that these workloads (i.e., those highlighted by blue squares, black triangles and brown circles) achieve optimal performance with one of the VP policies, indicating that their performance benefits accumulate to a certain degree due to the vertical partitioning on both cache and main memory banks. Shown in Figure 3 Quadrant I, different symbols denote workloads with different properties. For example, blue squares represent *A-VP friendly* workloads, which achieve the best performance with A-VP policy.

Quadrant IV contains workloads for which bank-only partitioning is beneficial while cache-only partitioning is detrimental. We study workloads in this quadrant and find that the performance benefits achieved by bank partitioning

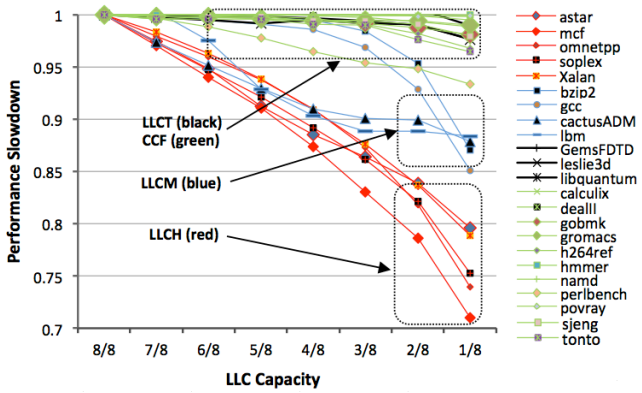


Figure 4. Normalized performance slowdown with different LLC capacity.

are largely offset by the side effect of cache partitioning (VP is not useful). Thus, for workloads in this quadrant, it is desirable to disable cache partitioning and enable bank partitioning. There are few workloads in Quadrant II and Quadrant III, indicating that bank-only partitioning does not bring negative impact under most of the cases.

3.3 Random-Interleaved Allocation (Multi-threaded)

Although the above analysis demonstrates that various partitioning policies achieve different levels of performance gains, there are cases where none of the partitioning-based memory allocation is preferred. One important scenario is that the running workload exhibits heavy data sharing, which defeats the purpose of any resource partitioning mechanism [16]. For example, multi-threaded workloads typically share considerable amount of data and thus multiple threads access the same memory or cache bank regardless whether the memory/cache is partitioned or not.

To optimize multi-threaded workloads, the recently proposed M^3 [18] enforces a random-interleaved page allocation to avoid hot spots and row buffer conflicts on heavily shared banks. We conducted experiments and verified that the random memory allocator outperforms partitioning-based approaches for multi-threaded workloads. Therefore, to handle multi-threaded workloads, we integrate a randomized page-interleaving scheme to achieve similar effects of M^3 in our framework (see Section 6.2).

An obvious conclusion can be drawn from the above presented quantitative study is that the effectiveness of a memory allocation policy depends on specific application characteristics, in particular the cache requirements. In practice, a workload could contain several simultaneously running applications with an arbitrary combination of diverse characteristics, making the task of determining an appropriate memory allocation challenging.

4. Application Classification

Determining an advantageous memory allocation policy requires an accurate prediction of a running workload's memory/cache characteristic and its reaction on each allocation policy. Based on our experiments we find that most multi-programmed workloads are not negatively affected by a modest bank-partitioning (<= 8 groups) scheme. By contrast, the performance of cache partitioning exhibits great variations due to different cache utilization behaviors of the running workloads (in Figure 3).

In order to verify the potential impact of cache utilization characteristics on cache partitioning policies, we collect performance slowdowns of various applications as the cache quota is reduced from 8/8 (entire cache is used) to 1/8. Each application is executed eight times and each time a different amount of LLC is assigned by the page-coloring based cache partitioning. Based on the results we classify applications' caching behaviors into four categories: Core Cache Fitting (CCF), LLC High (LLCH), LLC Middle (LLCM) and LLC Thrashing (LLCT). Figure 4 reports the classification of various benchmarks in the SPEC2006 benchmark suite [1]. CCF applications (denoted as green curves), such as *hmmer* and *namd*, do not degrade significantly when using fewer LLC resources since their working set sizes are small enough to fit into the L1 and L2 per-core private caches. LLCT applications (black curves), such as *libquantum*, are also insensitive to cache quotas, but due to cache thrashing behavior rather than small working set sizes. LLCH applications (red curves) such as *mcf* suffer the worst performance degradations from reduced cache quotas due to their resource hungry characteristics. Compared to LLCH, LLCM (blue curves) applications use fewer cache resource, thus the slowdowns are not as much as LLCH applications. For example, *gcc* and *bzip2* are LLCM as they suffer no significant degradation when cache decreases from 8/8 to 4/8. However, a sharp performance drop is observed when cache quota drops below 3/8.

4.1 Dynamic Classification

The static profiling approach used to generate Figure 4 requires multiple experiments for each application running and does not capture dynamically changing behavior. To predict cache requirement on the fly we explore the synergy between application page accesses and cache utilization. The key insight is that the number of *hot pages* (active pages used in a particular time interval and can be identified by the access bit [26,27] in the page table entry (PTE)) can reflect an application's LLC demand in many cases due to the DRAM row-buffer locality [19]. Figure 5 shows the correlation between number of hot pages and cache demands for several benchmarks. Taking *hmmer* as an example, the number of hot pages (denoted as red box) is extremely low (at most 19 hot pages over the entire sampling period). This indicates that a maximum amount of 19×4KB (4KB per page) cache resource is needed during the sampling period. Additionally, we test all benchmarks in SPECCPU 2006 in our experiments, but we only show several of them in Figure 5 due to the space problem.

To this end, a simple estimation of LLC utilization can be achieved by dividing the number of hot pages (NHP) by the number of pages the LLC can accommodate (NPC). This

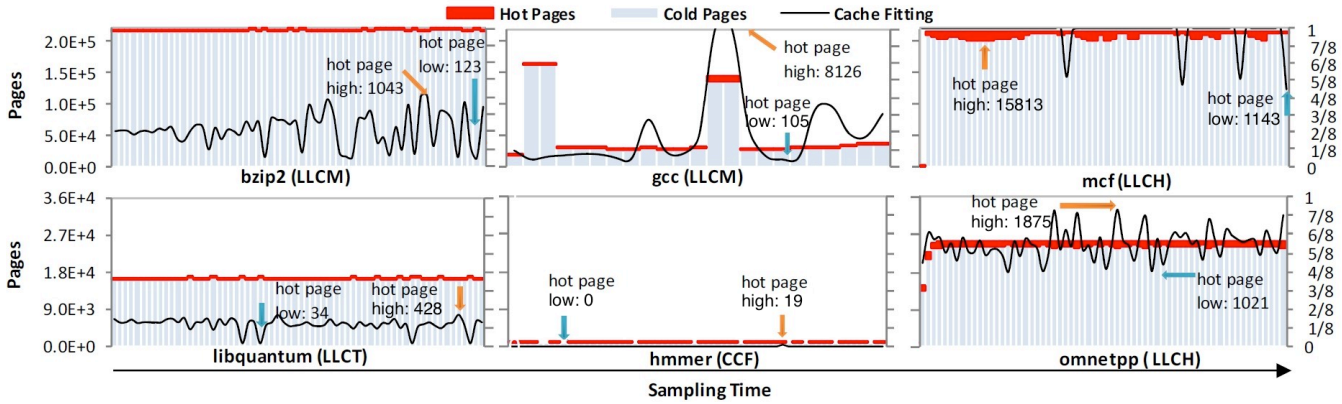


Figure 5. Applications' hot pages and their demands for LLC capacity.

metric (NHP/NPC) represents the percentage of LLC occupied by hot pages and is shown as the cache fitting curve in Figure 5 (vertical axis on the right side of each sub-figure). In the case of *hmmer*, less than 1/8 of the LLC is required, indicating a CCF classification. A sharp comparison is *mcf*, which is classified as LLCH since it has large amount of hot pages (1143 to 15813) that cannot be accommodated in most modern computers' LLC (e.g., 8M). For an LLCM application (e.g., *bzip2*), the number of hot pages falls between that of the LLCH and CCF application and the required LLC quota typically varies between 1/4 and 1/2. For an application that touches a large number of pages but exhibits very poor reference locality (i.e., some CCF applications such as *sjeng* visit many hot page but only a small amount of them are heavily accessed and benefit from being cached), using only the number of hot pages will mislead the above simple method to a wrong classification of LLCM or LLCH. To address this issue, we define *weighted page distribution* (WPD), a metric used to reflect page reference locality and can be obtained by per-page access counters (detailed in Section 4.2). Based on the above analyses, we devise an online application classification algorithm detailed next.

4.2 Classification Algorithm

Figure 6 illustrates the classification process where two tasks, JOB1 and JOB2, are launched in parallel. JOB1 is responsible for collecting the number of hot pages. Its sampling time interval is 3s in our system and the sampling duration in each interval is 10μs. Our experiments show that 10μs is enough to collect sufficient information while incurs negligible overhead. During each sampling JOB1 first clears *__access_bit* by the *pte_mkold()* kernel function, and then collects the number of hot pages (*__access_bit* set to 1) at the end of the sampling. Note that hot page numbers are averaged over several sampling intervals to avoid temporary spikes and reflect stable program behaviors.

JOB2 uses an array of page access counters to record the number of accesses for each page. Since the OS itself does not frequently reset the *__access_bit*, once set by the CPU, JOB2 employs a loop to periodically clear __access_bit and collects the access information during this period. JOB2 incurs slightly more overhead than JOB1, but the amortized overhead over the sampling time window (also 3s) is not high since it switches to the sleep mode after iterating 200 times (the time cost is far less than 3s). Based on the page access counters, JOB2 records the numbers of pages by grouping the counter values into five ranges: VH [150, 200], H [100, 150], M [64, 100], L [10, 64] and VL [1, 10]. For example, M denotes the number of pages with a counter value large than or equal to 64 but smaller than 100. Based on the above information, the WPD is computed as:

$$WPD = \frac{2\times VH + 1.5\times H + 1\times M + 0.5\times L + 0.1\times VL}{all\_used\_pages\_num},$$

where *all_used_pages_num* is the total number of pages accessed during a JOB2's sampling period (200 iterations). Moreover, we find that the total number of pages touched by an LLCT application (e.g., *libquantum* in Figure 5) does not show a great variation (<4% changes) during different execution time windows (i.e., after 200 iteration in every 3s) and it often generates more hot pages due to the memory intensive feature compared with CCF, making it easily to be identified. Based on the WPD metric to reflect reference locality, the hot page number and a series of thresholds, we devise a classification algorithm shown in Figure 6. The values of *ccf_threshold, hot_freq_threshold* and *llch_threshold* are 100, 10% and 1000, respectively.

The constants in our algorithm (i.e., sampling interval, weighted, thresholds, and etc.) are empirical values based on the analyses of all programs from SPECCPU 2006 with diverse memory features and a wide range of workloads combinations. Thus, we conclude that our approach is cost-effective, robust and may work well in many real cases. These values can be adjusted as necessary in the conditions of extreme environment changes.

5. Handling Multiplicity by Learning Rules

The application classification information obtained from the mechanism introduced above only reflects the partitioning preference for a single application, but the challenge of

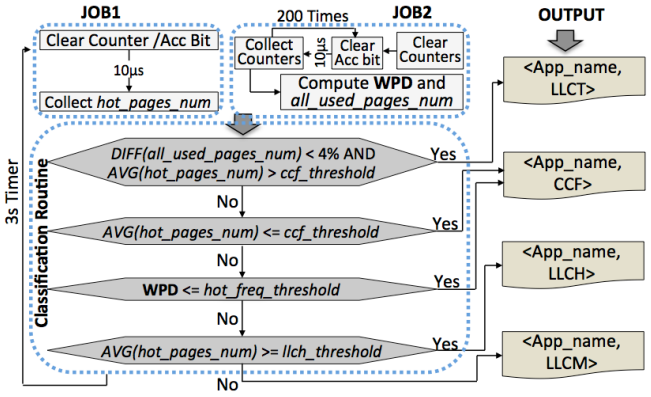


Figure 6. Online application classification algorithm. AVG(x) computes the average of x in three consecutive intervals and DIFF(x) returns the absolute value of the difference of x between two adjacent intervals.

selecting an appropriate scheme for co-running applications with an arbitrary combination of memory demands remains unaddressed. To tackle this challenge, we adopt a data mining approach to quantitatively study the impacts of various memory allocation schemes on over 2000 workloads. We summarize the outcome as a set of *partitioning rules* and *coalescing rules*, which can be used to handle diverse memory allocation needs for simultaneously running applications in multicore systems.

5.1 Partitioning Rules

Given a multi-programmed or multi-threaded workload, our first step is to select an appropriate memory allocation policy. To achieve this we collect a large amount of performance data from more than 10,000 experiments over 2000 workloads. For each workload, we use the framework introduced in Section 4 to obtain a *classification vector*, a notation to represent workload composition. For example, the classification vector of the workload {*libquantum, mcf, bzip2, hmmer*} is denoted as {<*lib*, LLCT>,<*mcf*, LLCH>, <*bzip2*, LLCM>, <*hmmer*, CCF>}. We run each workload with different policies and record the results as <cache-only: *x%*>, <bank-only: *y%*>, <A-VP: *z%*>, etc., where *x%*, *y%* and *z%* are performance improvements achieved by the corresponding policies. Based on the correlation between the classification vectors and the performance gains on different policies, we draw several interesting conclusions. First, almost all workloads that are combinations of LLCT and other type(s) of applications perform best on C-VP or A-VP. Second, a dominating percentage of workloads containing LLCH but not LLCT perform best on bank-only partitioning. Third, most workloads with LLCM but no LLCT or LLCH applications achieve best performance results with a modest cache partitioning scheme such as A-VP and B-VP. We summarize the above conclusions by the following three rules:

Rule-1: *Workloads containing LLCT and other applications (LLCH, LLCM, CCF) should use C-VP or A-VP (37.1% support, 94.4% confidence*[1]*).*

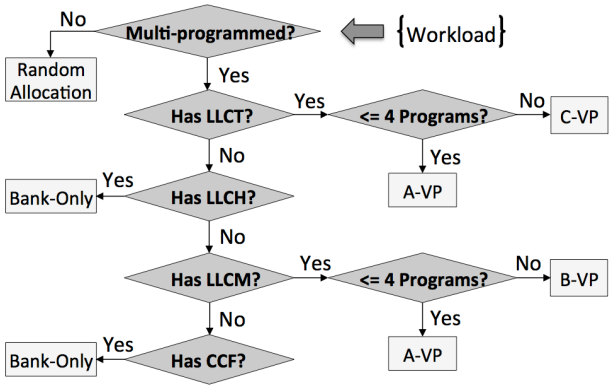


Figure 7. Memory allocation policy decision tree (PDT)

Rule-2: *Bank-only partitioning should be used for workloads with LLCH and LLCM applications but without LLCT applications (34.3% support, 83.3% confidence).*

Rule-3: *A-/B-VP should be used for 4-/8-programmed workloads with LLCM but no LLCT or LLCH applications (23.8% support, 87.9% confidence).*

The above analyses and rules also imply a priority in considering a memory allocation policy: LLCT > LLCH > LLCM > CCF. This ordering indicates that LLCT is the most "assaulting" type in that it brings negative impact for virtually all the other types of applications while CCF is the most susceptible classification and applications of this type hardly affect other applications' performance.

These results can be also explained by architecture knowledge. In particular, Rule-1 is likely to perform well on any LRU (least recently used)-based caches since LLCT applications are not well handled by the LRU policy [9] as they waste other applications' resource without being benefited. Rule-2 and Rule-3 can be also explained from the perspective of resource utilization.

For multi-threaded workloads, recent research [16] shows that bank partitioning only achieves slight performance gain. Additionally, Park et al. [18] argues that a random page-interleaved allocation scheme outperforms partitioning schemes. We conducted experiments for multi-threaded (see Section 7) workloads and verified their conclusion. Thus, we add another rule for multi-threaded workloads:

Rule-4: *Multi-threaded workloads should use random page allocation policy.*

Based on the four rules and their priorities relative to each other, we generate a memory management **policy decision tree (PDT)** shown in Figure 7. The PDT is useful for choosing appropriate policies for diverse workloads.

5.2 Coalescing Rules

Despite the advantage in eliminating interference, a pure partitioning based approach is not always preferable since it limits the cache capacity and can harm the performance for resource hungry applications (e.g., LLCH). To arrive at a middle ground between partitioning and sharing, we extend

[1] *Confidence* and *support* are terminologies in data mining. In our work *support* is defined as the proportion of workloads that contain the specific types of applications in a rule; *confidence* indicates the accuracy of that rule.

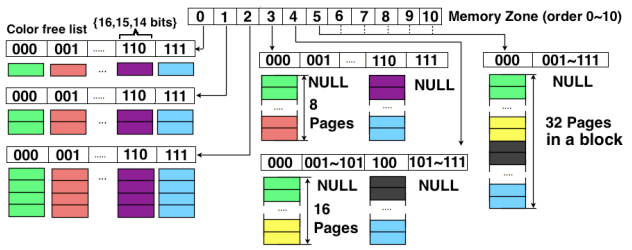


Figure 8. Free page list organization of Sub-system A in buddy system. Sub-system A organizes pages by 2 O-bits (bit 14,15) and one C-bit (bit 16). Thus, it contains 8 colors (000~111), and can facilitate searching pages for A/C-VP policies.

the partitioning decision tree with several coalescing rules that can be used to merge the partitioned resource quotas among certain types of applications.

We collect performance data of cache-only partitioning and represent the result for each workload as <(n×LLCT, m×LLCH, p×LLCM, q×CCF), *x%*>, where *n, m, p, q* are the numbers of applications of a certain type and *x%* is the performance gain achieved by the cache partitioning. Based on the results, we find that for almost all workloads that contain LLCH or LLCM but no LLCT applications (*n=0, m+p>0*) cache partitioning always hurts performance (*x%* < 0). Additionally, for the workloads containing only LLCT applications (*m=p=q=0, n>1*), the improvement is quite modest (*x%<1%*) and for CCF dominant workloads (*n=m=p=0, q>1*) no obvious impacts are observed. Further, we run multiple LLCT applications (lib.) together on 1/8 LLC capacity and find that the overall performance is similar to the cases where they are partitioned or share the entire cache. The same results are observed for CCF workloads. Thus, we derive the following coalescing rules:

Rule-5: *LLCH and LLCM applications should be coalesced together to share the partitioned colors and cache quota (support: 39.5%, confidence: 87.2%).*

Rule-6: *LLCT and CCF applications should be coalesced respectively to share the partitioned colors and small cache quota (support: 7.8%, confidence: 90.5%).*

The above coalescing rules are important complements to the partitioning rules for providing larger aggregate cache capacity and reducing misses under a partitioned cache. Coalescing might incur slightly higher bank contention among applications, but the benefits exceed the negative impact and HVR still performs well.

5.3 Combining Partitioning and Coalescing

In real production environments, applications can be launched and terminated arbitrarily. This dynamically changing workload composition can defeat any predetermined policy selection method. Combining and switching between partitioning and coalescing policies is particularly useful for handling dynamic changes in the running workloads. For example, several partitioned cache quotas under C-VP can be dynamically coalesced to form a larger aggregate quota for accommodating multiple non-

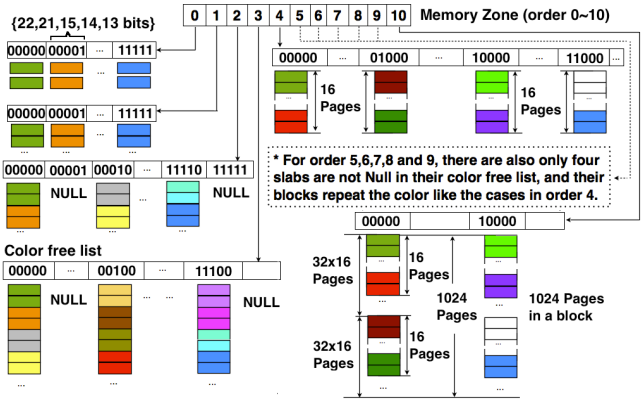


Figure 9. Free page list organization of Sub-system B. It organizes pages by bank bits 13, 14, 15, 21 and 22, and contains 32 colors, facilitating searching pages for B-VP policies, bank-level random or round-robin interleaving scheme.

conflicting applications launched in an arbitrary order (see Rule-5 and Rule-6). On the other hand, a coalesced space can be partitioned if an additional partition is needed when an assaulting application (e.g., LLCT) is launched.

6. Supporting HVR in Linux Kernel

The previously presented classification framework is implemented as kernel modules in the Linux kernel 2.6.32 in about 700 lines of source code. This section details our modification to the kernel data structures and paging algorithm to support the previously discussed policies and the partitioning/coalescing rules in a unified system. We implement HVR in roughly 3000 lines of source code over the existing kernel source tree. We use a page-coloring based scheme to re-organize all free physical pages. Our modified kernel maintains two page indexing systems: sub-system A and sub-system B. Sub-system A provides support for A/C-VP while sub-system B supports bank-only, B-VP and the random-interleaved page allocation policy. Note that the same page typically has different colors in different sub-systems. Section 6.3 introduces how sub-system A and B live and work together in our modified kernel. Based on the two sub-systems, we develop a hash-based searching algorithm (see Pseudocode 1) to allocate a page in O(1) time.

6.1 Page Indexing Sub-system A

The indexing sub-system A is illustrated in Figure 8. As Figure 8 shows, the Linux kernel buddy system maintains free physical pages in orders of blocks from 0 to 10. Each block in order-*n* contains 2^n continuous pages. The three bits used in sub-system A form a set of 8 colors (000~111), each of which has a free page list in our modified kernel.

In order-0 (upper left corner of Figure 8), each block is an individual page and the block list under a particular color is a set of pages of that color. For example, the block list under the green color in order-0 contains any free and non-continuous page with the three coloring bits being 000. Order-1 and order-2 are similar to order-0 except that two or four pages in a block are continuous. Each block in order-3

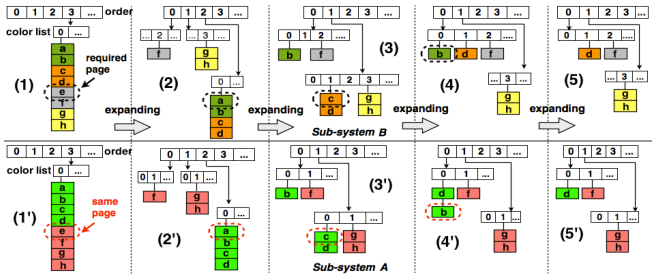


Figure 10. An example of free page list expanding in sub-system A and B

has 8 continuous pages and spans two colors since there is one coloring bit (bit 14) within the offset of a block in order-3. Similarly, each block in order-4 spans two coloring bits (bit 14~15) and thus has four colors (16 pages), as depicted in Figure 8. Since all the coloring bits are within a block offset starting from order-5, blocks of order-5 and above have in-block repeated 8-color (32 pages) patterns, as shown on the right of Figure 8.

Sub-system A supports C-VP since it uses the same bits (14~16) to organize the pages into different colors. It also supports A-VP. From the perspective of A-VP, the bit 16 is not a coloring bit thus the buddy system views two colors in sub-system A with the same lower two bits to be the same color (e.g., 111 and 011). The most significant coloring bit (bit 16) is simply not considered when choosing a color in the sub-system A page indexing structure. CASE 1 in Pseudocode 1 shows how to select a page of a given color from sub-system A.

6.2 Page Indexing Sub-system B

Sub-system B, shown in Figure 9, uses five bits (13~15, 21, 22), or 32 colors, to provides support for B-VP and bank-only partitioning, since all these bits denote the DRAM bank index. Within order-1, each block contains two consecutive pages of the same color (the addresses of these two pages only differ in bit 12). From order-2 to order-9 each block contains pages of more than one color since continuous pages in these orders spread across multiple colors. Order-10 is special in that each block spans 10 bits of the page address (bits 12~21) and contains 1024 continuous pages. Thus, all the binary combinations of the coloring bits 13, 14 and 15 form an 8-color 16-page group, which alternate as the upper non-coloring bits (bits 16~20) vary ($2^{20-16+1}$ = 32 groups). The bit 21 is also a coloring bit, and this repeats the pattern of the 8-color 16-page alternation with a different 8-color set. Thus, each block in order-10 has a pattern of 2 × 32 8-color 16-page groups. The bit 22, the lowest block address in order-10 and also a coloring bit, repeats the above 2 × 32 group pattern with an entirely different 16-color set in a different block. To choose a page with a particular color in a certain order, we use the algorithm shown in CASE 2 in Pseudocode 1.

In particular, in sub-system B, the random-interleaved page allocation for **multi-threaded** workloads can be easily achieved by randomly selecting pages in the order-0 free list.

Pseudocode 1: Hashing algorithm for selecting pages
Input: (1) order; (2) target_color **Output:** one page of target color

```
BEGIN
/*CASE 1: Physical pages organized based on bits 14~16*/
  IF using 14,15,16 bits THEN
    SWITCH (order)
```

case	0~2	3	4	5~10
colors_per_block =	1	2	4	8

```
    END SWITCH
    block_color = (target_color / colors_per_block) × colors_per_block;
    page_index = (target_color - block_color) × 4;
  END IF

/*CASE 2: Physical pages organized based on bits 13~ 15, 21~22*/
  IF using 13, 14, 15, 21, 22 THEN
    SWITCH (order)
```

case	0~1	2	3	4~9	10
colors_per_block =	1	2	4	8	16

```
    END SWITCH
    block_color = (target_color / colors_per_block) × colors_per_block;
    IF order is 10 AND the color bits are x1xxx THEN //The 4th bit is 1
      page_index = (target_color - block_color - 8) × 2 + (1 << 9);
      // As shown in Figure 9: 32 blocks × 8 colors = 1024 blocks
      ELSE  page_index = (target_color - block_color) × 2;
    END IF
  END IF
  Expand color block (page_index, order)
  // physical pages represented by "struct page" are in page[] array
  // in Linux kernel.
  RETURN page[page_index] and remove this page from free list.
END
```

* target_color is the color of the requested page.
* block_color is the color of the first page in a block.
* colors_per_block is the number of colors in a block.

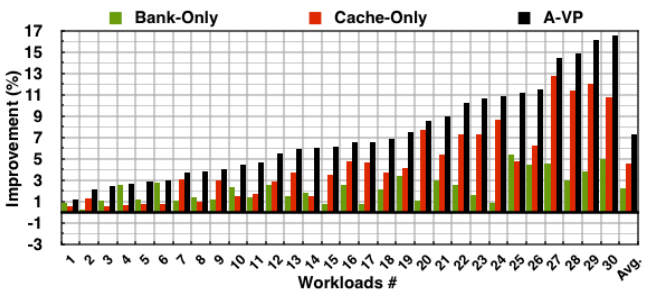
Moreover, Similar effect in M^3 [18] can be also achieved by allocating physical pages with the 32 colors (00000~11111) in a round-robin fashion and interleaving the required pages evenly across all banks to reduce the potential bank conflicts. Therefore, our framework, HVR, is able to support both multi-programmed and multi-threaded workloads.

6.3 Consistency of Sub-systems

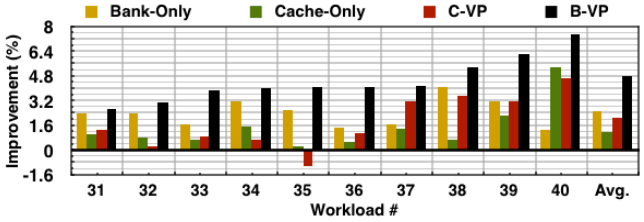
When a page is requested and removed from a free block list in one of the two sub-systems, the corresponding entry in the other sub-system should also be removed to keep the consistency between the two sub-systems.

Given a color, the speed of searching its corresponding free block list is vital and the operation should be low-overhead. By default, a free page request is serviced by order-0 if the free block lists under the requested color in order-0 is not empty. When order-0 cannot satisfy a request, block lists of higher orders are searched. In this scenario, continuous pages are broken into lower order blocks (fewer continuous pages), which will be inserted into the free block lists of appropriate orders. This process is known as *expand* in the kernel, shown in Figure 10.

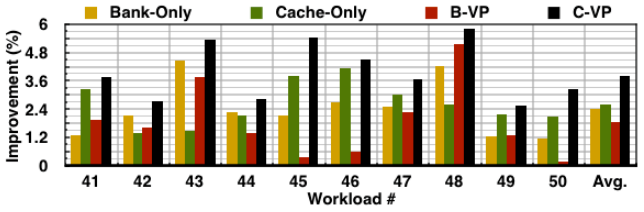
Suppose a thread in sub-system B requires a page of gray color and there is no gray page in order-0, 1 and 2, as shown in Figure 10 (1). The system has to expand a large block of 8 consecutive pages in order-3 into one order-2 block (green and orange), one order-1 block (yellow) and one order-0



(a) 4-programmed workloads (A-VP performs best) .



(b) 8-programmed workloads (B-VP performs best).



(c) 8-programmed workloads (C-VP performs best).

Figure 11. Performance gains of A/B/C-VP.

block (gray), in addition to the gray page to service the request. This is illustrated in Figure 10 (2). Since the physical page needs to be removed in the other sub-system, the expand process is triggered for sub-system A to remove a red page in order-3, as shown in Figure 10 (2'). The red page in sub-system A and the gray page in sub-system B are associated with the same physical page. The following steps (3) ~ (5) and (3') ~ (5') in Figure 10 show similar expand processes as different pages are requested and how the two sub-systems keep in sync with each other.

6.4 Implementation Complexity

Although the entire mechanism seems complicated, it is relatively simple in implementation and efficient in performance. In memory management module in Linux kernel, each physical page is represented by a *page* structure and linked into the buddy system by a *list_head* structure based on the *order* member in the *page* structure. We add two additional *list_head* structures, *lruA* and *lruB,* to track locations of pages in sub-system A and sub-system B, respectively. Every *page* structure is linked to sub-system A by *lruA* and to sub-system B by *lruB*. Each *list_head* item in the two sub-systems contains a pointer to each other so that the two sub- systems can be synchronized in O(1) time. The color(s) assigned to a particular process is maintained by a *color mask* added to the *task_struct* structure and used in the O(1) hash searching (Pseudocode 1) for page allocation.

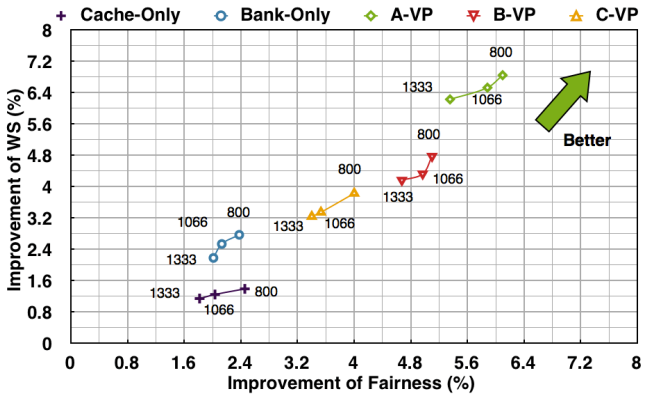


Figure 12. Performance of different policies as bandwidth changes (the baseline is the unmodified Linux kernel).

6.5 Coalescing and Page Migration

The coalescing rules (Section 5.2) can be implemented by assigning the same color mask to multiple applications (e.g., two CCF applications). When two or more applications are coalesced, the color masks in their *task_struct* are identical so that they share the same cache memory quota.

A common issue associated with coalescing is that page migration typically needs to be involved, which is expensive, especially when invoked frequently. Migration is also needed for application re-classification. To mitigate page migration overhead we always enable bank-only partitioning at the very beginning. Doing so rarely brings negative impact (see Section 3.2) and avoids page migrations due to a transition from a non-bank partitioning to a bank partitioning policy. Our experiments also show that by enabling bank-only partitioning as a baseline, migrations can be greatly reduced upon transitions between any two policies (with bank partitioning). Moreover, we use lazy migration [15] in our framework to avoid unnecessary migration. Doing these in practice greatly reduces the number of pages need to be migrated (see Figure 16), and thus reduce the overhead.

7. Evaluations

7.1 Experimental Methodology

Our experimental machine has a quad-core eight-thread 2.8GHz Intel i7-860 processor with 8MB 16-way LLC and 8GB 64-bank DDR3 main memory. The machine runs CentOS Linux 5.4 with the kernel 2.6.32. We use SPEC CPU2006 suite [1] for multi-programmed workloads and PARSEC benchmark suite [2] for multi-threaded workloads. All programs are compiled by *gcc* 4.4.3 with the -O3 optimizations. Similar to previous work, we use *weighted speedup* [12] (WS) to measure system performance and *maximum slowdown* (MS) [12] for fairness:

$$Weighted\ Speedup\ (WS) = \sum \frac{Runtime_{along}}{Runtime_{together}}$$

$$Maximum\ Slowdown\ (MS) = Max\left\{\frac{Runtime_{together}}{Runtime_{alone}}\right\}$$

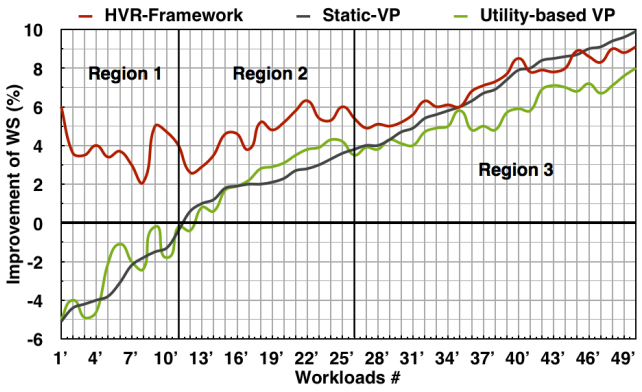


Figure 13. Performance improvements of different schemes.

We compare several memory allocation schemes including the unmodified paging system in the Linux kernel, utility-based partitioning [15], DRAM bank partitioning [16], random allocation [18] and our proposed HVR system.

7.2 Impact of VP Policies

Figure 11 shows that for workloads that benefit from cache-only and bank-only partitioning (50 workloads in Quadrant I of Figure 3), VP can accumulate the performance gains. For these workloads, all A-/B-/C-VP achieve the optimal performance. For instance, A-VP can achieve up to 16.7% improvement over the baseline system, while 5.9% and 11.7% over the cache-only and bank-only partitioning, respectively. We also try the random-interleaved page allocation for these multi-programmed workloads, and it performs worse than the partitioning policies. Figure 12 summarizes the performance and fairness improvements of various policies based on an average of workloads in our experiments. To demonstrate that the proposed scheme performs robustly under different memory bandwidth, we change the memory frequency from 1333 to 800MHz. Figure 12 illustrates that on average all the three vertical partitioning policies outperform the horizontal cache-only or bank-only partitioning schemes. Particularly, A-VP achieves a nearly 6% improvement over the cache-only partitioning and a 5% gain over the bank-only partitioning. As bandwidth decreases, the contention becomes more severe and the three vertical policies can bring even larger improvements. Therefore, we can draw conclusion that vertical partitioning brings additional benefits over horizontal partitioning and is a promising memory management mechanism for future multicore systems with increasing bandwidth pressure.

7.3 Overall Performance

Section 7.2 shows that using the proposed VP policies can bring performance gains over the unmodified kernel memory allocation and prior schemes. This section reports the effectiveness of HVR framework, which utilizes these VP policies more flexibly based on the application classification, partitioning and coalescing rules, to handle diverse and dynamically changing workload characteristics and behaviors in daily computing environments.

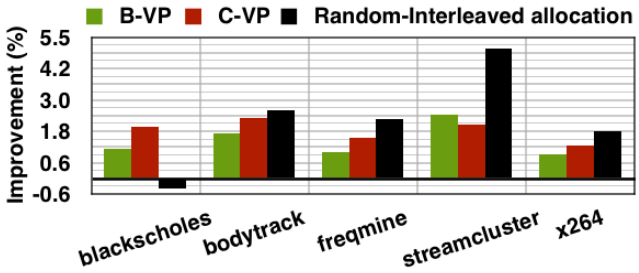


Figure 14. Performance of multi-threaded workloads.

7.3.1 Dynamic Policy Selection

Our HVR framework can automatically select appropriate memory allocation policies by collecting applications' characteristics and searching a policy in the policy decision tree. To demonstrate the superiority of HVR over the static partitioning and prior utility-based partitioning [15] approach we compare three mechanisms with the baseline system. Static vertical partitioning (SVP) adopts A-VP for 4-programmed workloads and B/C-VP for 8-programmed workloads. Utility-based VP (UVP) dynamically adjusts cache partitioning based on cache misses monitored through performance counters. Figure 13 reports the performance for the three schemes over 50 randomly generated workloads sorted by their performance improvements achieved by SVP. In region 1, both SVP and UVP achieve negative performance gain (up to -5.0%) over the baseline. In contrast, HVR improves performance by up to 6.1% over the baseline and 11% over SVP and UVP. A careful analysis reveals that workloads in region 1 are primarily LLCH dominate workloads, for which the cache partitioning is detrimental. Thus, SVP and UVP policies are not suited for these workloads. HVR achieves gains by automatically identifying workload characteristics and selecting the bank-only partitioning for these workloads.

In region 2, most workloads are 8-programmed ones with LLCT applications. HVR outperforms SVP and UVP due to resource coalescing. For instance, the workload 22' contains 5 LLCT, 2 LLCH and 1 LLCM applications. HVR maps all LLCT applications to 1/8 cache, thus the rest 7/8 cache quotas are shared by LLCM and LLCH applications, contributing to the improvement of system performance.

In region 3, in most cases, HVR also outperforms other two approaches, since HVR is capable of selecting proper VP policies and using coalescing rules. But for some workloads, SVP (Static VP) performs slightly better (0.4% better than HVR on average). By looking into workloads in this region we find a high percentage of A/B-VP friendly workloads containing multiple LLCM applications. Since an LLCM application requires modest cache capacity and typically maintains a steady rate of cache utilization, the SVP avoids dynamic overhead as it determines the partitioning policy only once at the very beginning by offline profiling. By contrast, dynamic utility-based approaches incur non-negligible overhead [15,26] that may offset the partitioning gains due to expensive page migrations induced by page re-coloring and performance

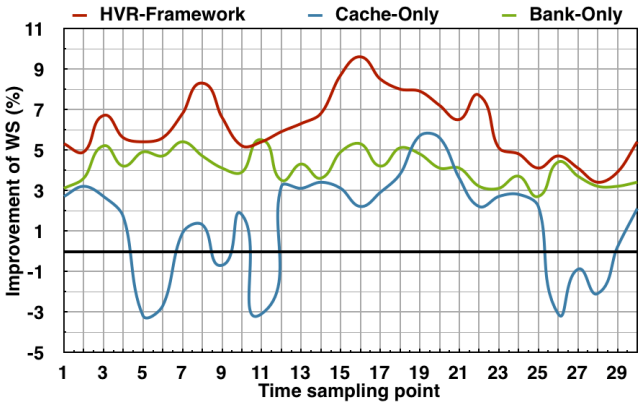


Figure 15. Real-time performance of HVR framework.

counter penalty. But, fortunately, HVR avoids offline profiling and will not incur significant overhead due to the page-table-based lightweight online profiling and the stable classification approach. More details about overhead are discussed in Section 7.4.

As previously mentioned, performance benefits can be achieved for multi-threaded workloads through adopting a random-interleaved page allocation approach. In Figure 14 our experimental results show that the random-interleaved page allocation policy outperforms B/C-VP policies for various 8-threaded workloads. HVR supports random-interleaved page allocation (see Section 5, Section 6.2, and Figure 7), which is automatically selected for multi-threaded workload based on the policy decision tree.

7.3.2 Performance for real-time Changing Workloads

Figure 15 reports real-time performance captured through Intel processor's IPC performance counters for cache-partitioning, bank-partitioning and HVR. During the entire testing span we inject applications of various kinds at different times. In the meantime, previously launched application may terminate upon completion. From the figure we can see the performance of cache partitioning fluctuates between 6% and -3%. At sampling time points 5,6, 9, 11, 26, 27 and 28 the performance of cache partitioning drops below 0. This is because at these points LLCH applications are launched and the performance degrades due to limited cache resources. HVR avoids this degradation by identifying these LLCH applications and then coalescing resources for them. At time 16, HVR achieves peak IPC performance improvement. This is the point where LLCT, LLCH, LLCM and CCF applications are all running and they are appropriately segregated to eliminate perturbation while some of them are shared to use larger amount of resources.

Compared to HVR, bank-only and cache-only partitioning approaches achieve modest gains (5% and 7% worse than HVR, respectively) due to the inability of selecting policies and coalescing resources dynamically. Note that bank-only partitioning achieves relatively stable performance over time and this trend is consistent with our conclusion in Section 3 (see Figure 3).

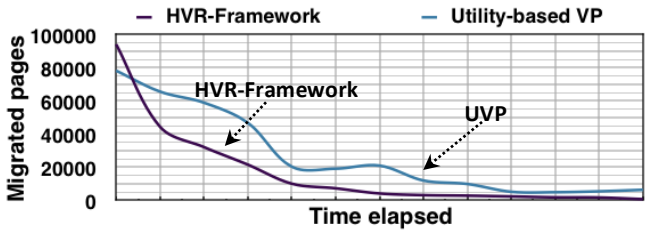


Figure 16. Number of page migration operations.

7.4 Overhead and Discussion

Overhead of HVR comes from the following three sources: **(1).** Page table sampling of JOB1 and JOB2 in the workload classification process. The costs of page table traversal depend on application's memory footprint. In our experiments, the time for page table traversal ranges from 5µs (povray) to 4.46ms (mcf). Thus, the amortized overhead of JOB1 and JOB2 are negligible. Moreover, JOB2's sampling interval grows with an increasing step once it collects sufficient information to complete the initial classification process, and thus its overhead is further reduced for long running workloads. In the worst case, JOB2 only brings 0.6% overhead. For workloads with extremely large memory footprint, a random sampling can be adopted for a tradeoff between the sampling overhead and classification accuracy.

(2). The page indexing in the modified buddy system. As our page searching routine can allocate a page in O(1) time and the synchronization between the two sub-systems is also highly efficient, our modified kernel does not bring obvious overheads (< 0.3% on average) during page allocation.

(3). Page migrations caused by re-coloring in dynamic policy adjustment. Migrating a 4KB page costs around 3µs on our platform. Fortunately, our approach does not incur too many page migrations because it relies on stable classification information that typically changes only when an application starts or terminates. In our experiments, an extreme case requires up to 400MB (100,000 pages) to be migrated, and the time cost is around 30s. However, the entire workload runs for more than 30 minutes and thus the overhead is 1.7% at most. Moreover, since our mechanism uses the lazy page migration [15] that only migrates a page when necessary, the average overhead is less than 0.8%. Figure 16 shows that under the extreme case our mechanism migrates fewer pages than the utility-based approach, in which the number of migrated pages fluctuate over time and incurs higher migration overhead in practice.

8. Related Work

There is a large body of related work on cache and memory allocation and partitioning. At the main memory level, Prashanth *et al.* [29] proposes DRAM channel partitioning that requires hardware and system modifications to segregate data from different threads into different channels to eliminate interference. Park *et al.* [18] adopt a random allocation algorithm to scatter allocated pages to multiple banks to avoid conflicts for multithreaded workloads. Liu et

al. [16] use page-coloring to partition DRAM banks to avoid contention of multiple programs. Kaseridis *et al.* [11] propose bandwidth-aware memory sub-system management for avoiding resource contention. Various approaches are also proposed to manage LLC [5,8,14,15,20,31,32,33,34]. In particular, Qureshi *et al.* [32] design a utility-based cache partitioning scheme that allocates appropriate cache resources based on application miss rate monitored through dedicated hardware. More recently, cache partitioning is also adopted in heterogeneous GPU-CPU architectures to promote fair resource sharing among CPU and GPU applications [30], which exhibit drastically different memory access characteristics. Other efforts [3,9,12,15,25,28] classify workloads based on hardware profiling, and then choose appropriate scheduling policies for different classifications. OS-level approaches for memory utilization monitoring [6,7,26,27] have also been studied to provide knowledge for resource management.

9. Conclusions

We propose and implement a practical, unified, and efficient multi-policy memory management framework named HVR to address the challenge of allocating appropriate memory resources for modern diverse applications. HVR seamlessly integrates several existing schemes and new vertical partitioning policies by leveraging O-bits and the page coloring technique. Through a quantitative study on a large quantity of experiments we verify that HVR can automatically select appropriate policies based on application needs and achieve over 10% performance benefits compared to prior allocation methods in many cases.

Acknowledgments

We thank the reviews for their feedback. Lei Liu and Chengyong Wu are supported by the 863 Program under grant No. 2012AA010902, and 973 Program under grant No. 2011CB302504; the National Natural Science Foundation (NSF) of China under grants No. 60873057, 60921002, 60925009, 61033009 and 61202055. Zehan Cui, Yungang Bao and Mingyu Chen are supported by the 973 Program under grant No. 2011CB302502; NSF under grants No. 60903046, 61221062, 60925009 and 61331008.

References

[1] Standard Performance Evaluation Corporation. Available from: http://www.spec.org/cpu2006/CINT2006/.

[2] C. Bienia, S. Kumar, J.P. Singh, and K. Li. The PARSEC benchmark suite: Characterization and architectural implications. in *PACT*. 2008.

[3] D. Chandra, F. Guo, S. Kim, and Y. Solihin. Predicting Inter-Thread Cache Contention on a Chip Multi-Processor Architecture. in *HPCA*. 2005.

[4] S. Cho and L. Jin. Managing distributed, shared L2 caches through OS-level page allocation. in *MICRO*. 2006.

[5] H. Cook, M. Moreto, S. Bird, K. Dao, D.A. Patterson, and K. Asanovic, A hardware evaluation of cache partitioning to improve utilization and energy-efficiency while preserving responsiveness. in *ISCA*. 2013

[6] P.J. Denning, The Working Set Model for Program Behaviour. Commun. ACM, 1968. **11**(5).

[7] X. Ding, K. Wang, and X. Zhang. SRM-buffer: an OS buffer management technique to prevent last level cache from thrashing in multicores. in *EuroSys*. 2011.

[8] X. Ding, K. Wang, and X. Zhang. ULCC: a user-level facility for optimizing shared cache performance on multicores in *PPoPP*. 2011.

[9] A. Jaleel, H.H. Najaf-Abadi, S. Subramaniam, S.C. Steely, and J. Emer, CRUISE: cache replacement and utility-aware scheduling, in *ASPLOS*.2012.

[10] M.K. Jeong, D.H. Yoon, D. Sunwoo, M. Sullivan, I. Lee, and M. Erez. Balancing DRAM locality and parallelism in shared memory CMP systems. in *HPCA*. 2012.

[11] D. Kaseridis, J. Stuecheli, J. Chen, and L.K. John. A bandwidth-aware memory-subsystem resource management using non-invasive resource profilers for large cmp systems. in *HPCA*. 2010.

[12] Y. Kim, M. Papamichael, O. Mutlu, and M. Harchol-Balter. Thread cluster memory scheduling: Exploiting differences in memory access behavior. in *MICRO*. 2010.

[13] K.C. Knowlton, A Fast storage allocator. Communications of the ACM, 1996.

[14] J. Liedtke, H. Hartig, and M. Hohmuth. OS-controlled cache predictability for real-time systems. in *RTAS*. 1997.

[15] J. Lin, Q. Lu, X. Ding, Z. Zhang, X. Zhang, and P. Sadayappan. Gaining insights into multicore cache partitioning: Bridging the gap between simulation and real systems. in *HPCA*. 2008.

[16] L. Liu, Z. Cui, M. Xing, Y. Bao, M. Chen, and C. Wu. A software memory partition approach for eliminating bank-level interference in multicore systems. in *PACT*. 2012.

[17] W. Mi, X. Feng, J. Xue, and Y. Jia. Software-hardware cooperative DRAM bank partitioning for chip multiprocessors. in *NPC*. 2010.

[18] H. Park, S. Baek, J. Choi, D. Lee, and S.H. Noh, Regularities considered harmful: forcing randomness to memory accesses to reduce row buffer conflicts for multi-core, multi-bank systems, in *ASPLOS*. 2013.

[19] S. Rixner, W.J. Dally, U.J. Kapasi, P. Mattson, and J.D. Owens. Memory access scheduling. in *ISCA*. 2000.

[20] T. Sherwood, B. Calder, and J. Emer. Reducing cache misses using hardware and software page placement. in *ICS*. 1999.

[21] L. Soares, D. Tam, and M. Stumm. Reducing the harmful effects of last-level cache polluters with an OS-level, software-only pollute buffer. in *MICRO*. 2008.

[22] D. Tam, R. Azimi, L. Soares, and M. Stumm. Managing shared L2 caches on multicore systems in software. in *WIOSCA*. 2007.

[23] A.S. Tanenbaum, Modern Operating Systems. 3rd ed. 2008: Pearson-Prentice Hall.

[24] A. Wolfe. Software-based cache partitioning for real-time applications. in *RCS*. 1993.

[25] Y. Xie and G. Loh. Dynamic classification of program memory behaviors in CMPs. in *the 2nd Workshop on Chip Multiprocessor Memory Systems and Interconnects*. 2008.

[26] X. Zhang, S. Dwarkadas, and K. Shen. Towards practical page coloring-based multicore cache management. in *EuroSys*. 2009.

[27] P. Zhou, V. Pandey, J. Sundaresan, A. Raghuraman, Y. Zhou, and S. Kumar. Dynamic tracking of page miss ratio curve for memory management. in *ASPLOS*. 2004.

[28] S. Zhuravlev, S. Blagodurov, and A. Fedorova. Addressing shared resource contention in multicore processors via scheduling. in *ASPLOS*. 2010.

[29] Muralidhara, S. Prashanth, L. Subramanian, O. Mutlu, M. Kandemir, and T. Moscibroda. Reducing memory interference in multicore systems via application-aware memory channel partitioning. in *MICRO* 2011.

[30] J. Lee and H. Kim. TAP: A TLP-aware cache management policy for a CPU-GPU heterogeneous architecture. in *HPCA*. 2012.

[31] S. Kim, D. Chandra, and Y. Solihin. Fair cache sharing and partitioning in a chip multiprocessor architecture. in *PACT*. 2004.

[32] M. K. Qureshi and Y. N. Patt. Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches. in *MICRO*. 2006.

[33] S. Srikantaiah, M. Kandemir, and Q. Wang. Sharp control: controlled shared cache management in chip multiprocessors. in *MICRO*. 2009.

[34] Y. Xie and G. H. Loh. Scalable shared-cache management by containing thrashing workloads. in *HiPEAC*. 2010.